Justin Liang

ECE 156A

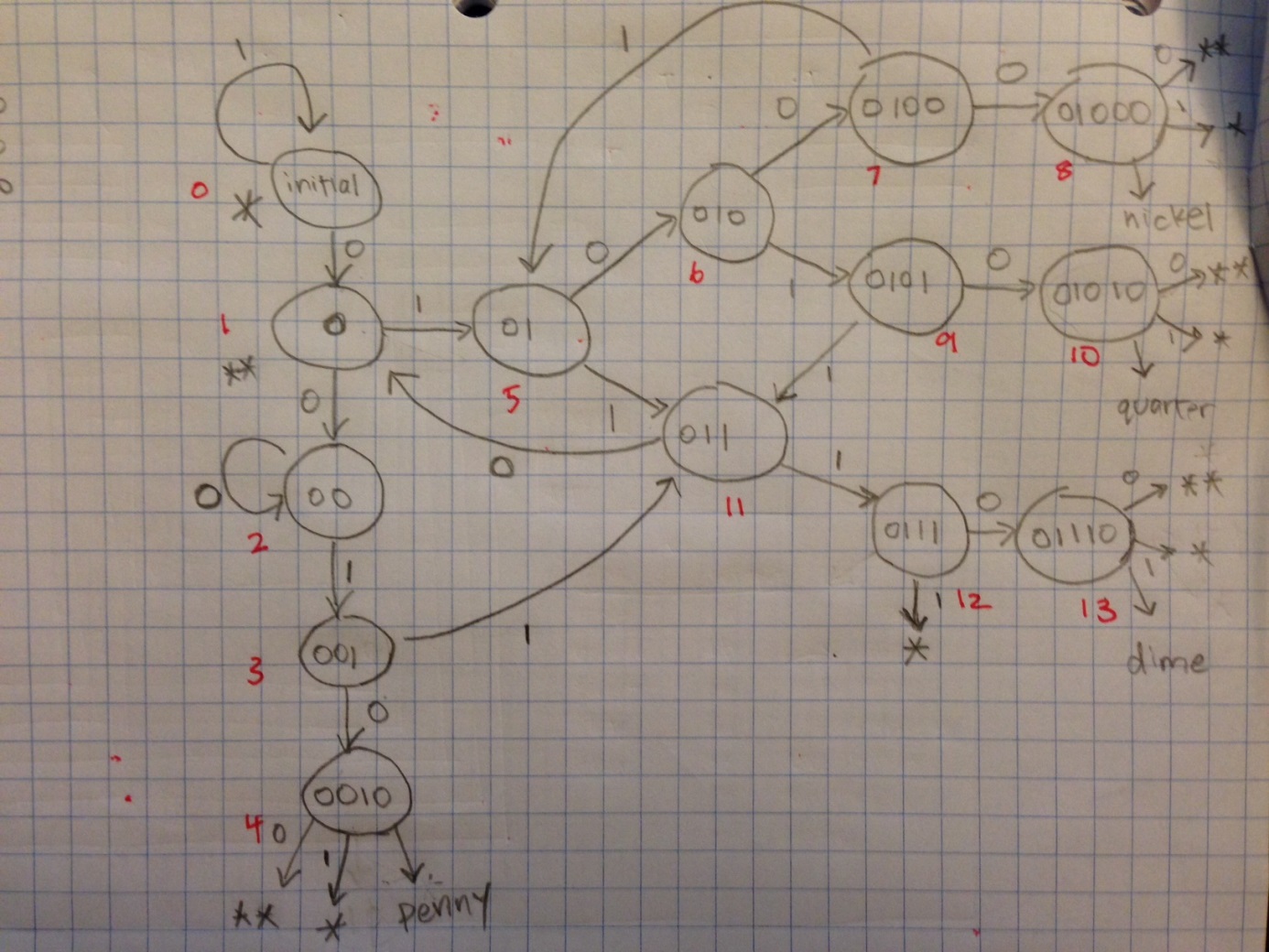
Due: 11/20/13

Homework #4

**Problem 1:**

coinSensor:

In this module, I constructed a finite state machine to model the coin sensor’s behavior. Coin sequences are reversed in the waveform because bits are read right to left instead of left to right. After I constructed the state diagram, the Verilog code is modeled after the state diagram using case statements. The reset signal should be omitted, because it does nothing. Below is my state diagram:



piggyBank:

In this module, I simply implemented an 8-bit register that stores the credit of the user. The 8-bit register is initialized to $0.00 (00000000). The Verilog code is simply if/elseif statements that appropriately add or subtract values every time a coin or a fruit is asserted. The module outputs the 8-bit register *credit*. When the reset signal is asserted, the 8-bit register is set to $0.00 (00000000).

purchaseMngr:

In this module, I implemented the purchase manager using simple if/elseif statements. Inside an always block, I first check to see if a buy signal is asserted. If a buy signal is asserted, I check to see what product number is asserted. After checking what product number is asserted, I output the appropriate product signal if the cost of that product is less than the current credit stored. If the cost of the product is more than the current credit stored, an error signal is outputted. If a buy signal is not asserted, all product and error signals are not asserted no matter what product signal is asserted.

sevenSegDispMngr:

In this module, the implementation got a little more complex than the other modules. First, I created 5 new signals called astart, bstart, cstart, dstart, and estart. When these signals are asserted, dig1 and dig0 will output the appropriate product/error letters (aa, bb, cc, dd, ee).

Next, I created a new type of down counter that starts from 4 and counts to 0 in an always block. When it reaches 0, all start signals are set to 0 and downStart is set to 0 as well to stop the counter from counting. The reason that it counts from 4 to 0 is because counting from 6 to 0 gave me 8 clock cycles to display the product/error letters. Counting from 5 to 0 gave me 7 clock cycles to display the product/error letters, because the counter stops counting one cycle later than expected. Therefore, counting from 4 to 0 guaranteed expected functionality.

Next, I have a separate always block, that detects apple, banana, carrot, date, or error signals. When one of these signals is detected, the appropriate start signal to signal the output to keep displaying the appropriate letters. The downReset and downStart signals are asserted as well to reset the down signal and start the down signal.

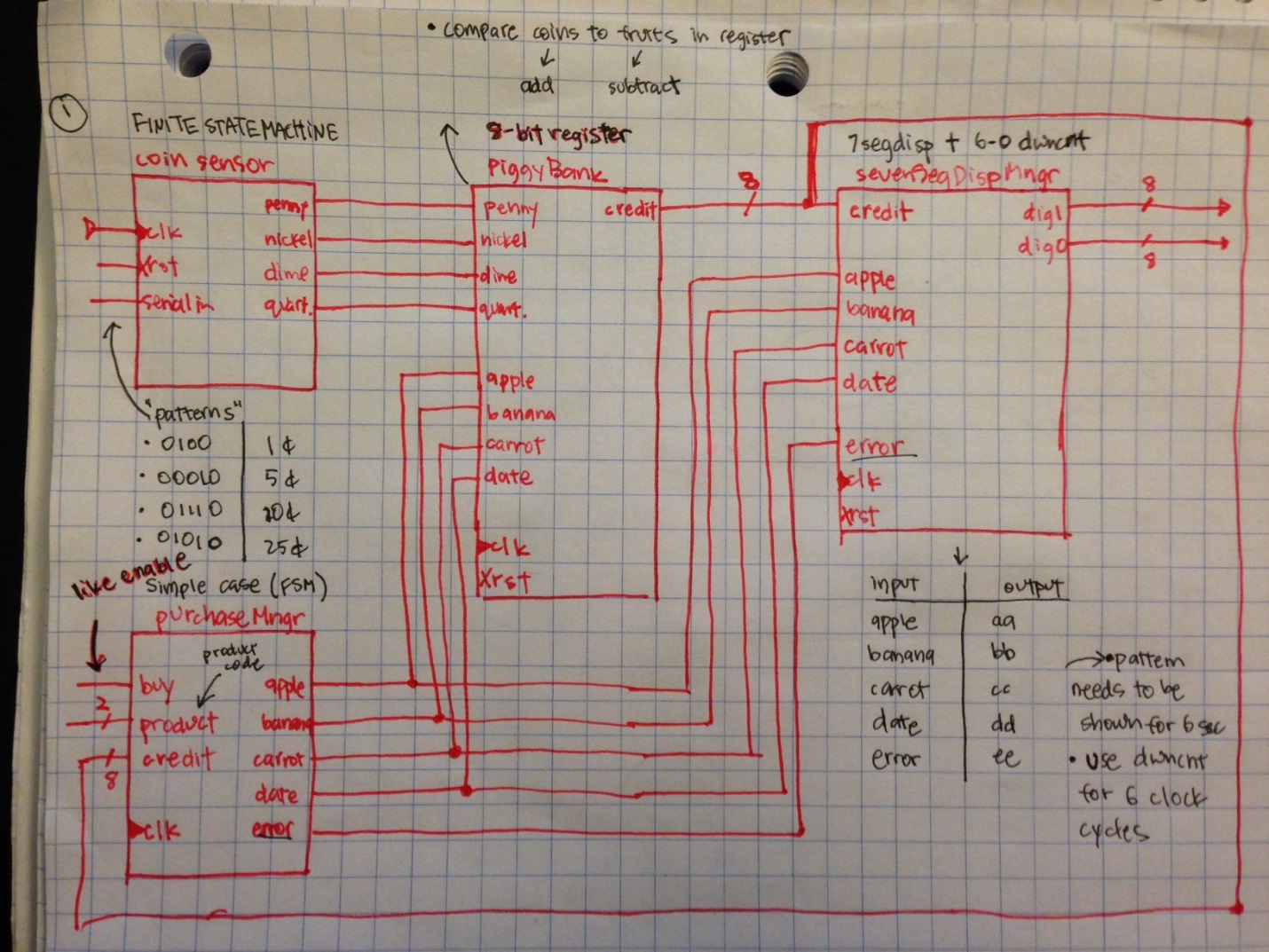
In my last always block, it detects whether a start signal is asserted. If it is asserted, it will make din1 and din0 output the appropriate letters.

Two bcdto7 modules from the previous homework are implemented to decode binary signals to the appropriate hexadecimal display signals. Below is the following table for the bcdto7 display.

|  |  |  |
| --- | --- | --- |
| Input | Output | Display |
| 0000 | 1111110 | 0 |
| 0001 | 1100000 | 1 |
| 0010 | 1101101 | 2 |
| 0011 | 1111001 | 3 |
| 0100 | 0110011 | 4 |
| 0101 | 1011011 | 5 |
| 0110 | 1011111 | 6 |
| 0111 | 1110000 | 7 |
| 1000 | 1111111 | 8 |
| 1001 | 1111011 | 9 |
| 1010 | 1110111 | A |
| 1011 | 0011111 | B |
| 1100 | 1001110 | C |
| 1101 | 0111101 | D |
| 1110 | 1001111 | E |
| 1111 | 1000111 | F |

vendingMachine:

In this module, I simply connected all 4 existing modules together and assigned appropriate wires to their inputs and outputs. Below is the following schematic:

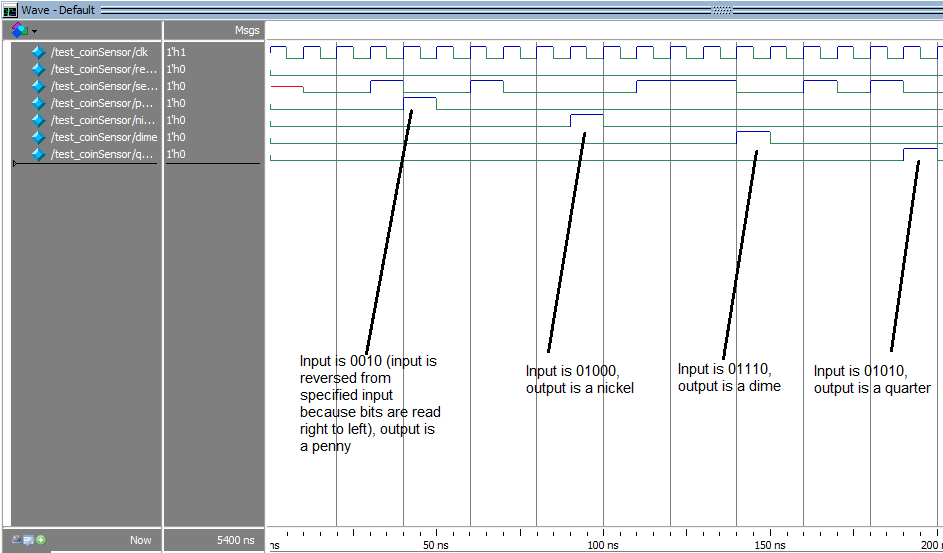


**Problem 2:**

For verification, I tested functionality of each module separately before testing functionality of the whole circuit. By doing this, I can pinpoint bugs quickly because of a smaller design. Once I obtained correct functionality for each individual component, it was a lot easier to verify correctness for the whole circuit.

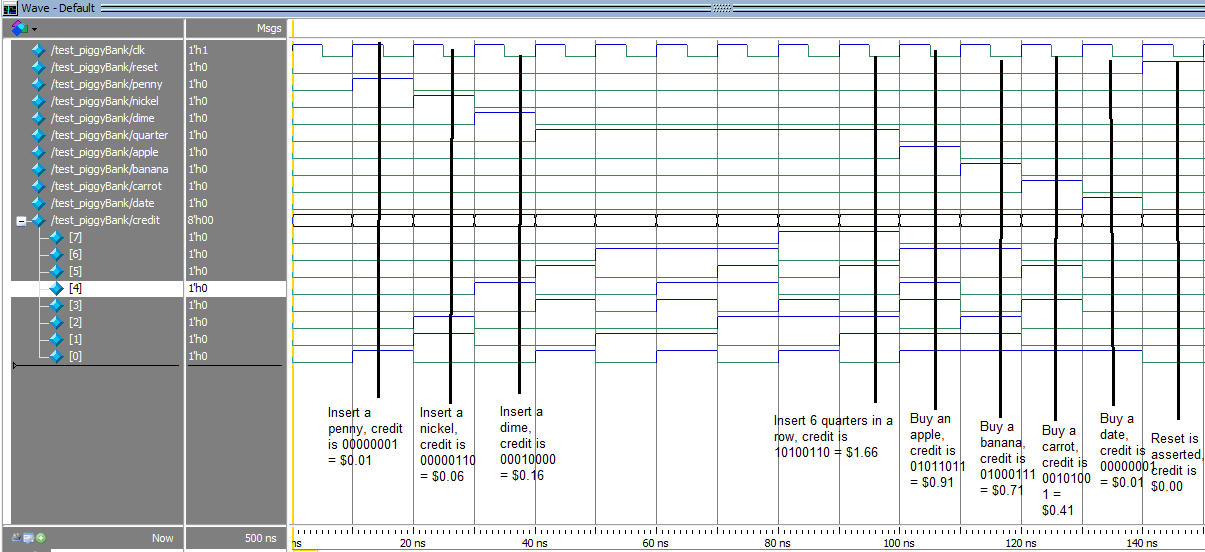
coinSensor:

First I tested the coinSensor module. I provided appropriate input patterns to serialIn to see if the correct coin signal will be outputted. This module was fairly easy to test as there are only 4 cases. The reset signal was omitted in testing because I did not think it was necessary to have a reset signal for the coin sensor in the first place (what are we resetting?). Below is the waveform for coinSensor.



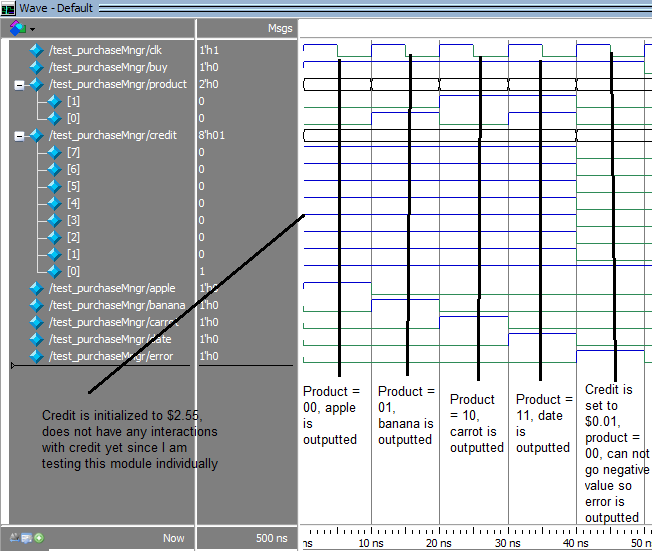
piggyBank:

To test this piggyBank, I inserted each coin once to see if the correct credit is outputted. First, I inserted a penny, then a nickel, then a dime, then a quarter, then 5 more quarters to get a total of $1.66. You can see on the waveform below that the correct credit is outputted with each coin insertion. Next, I tested to see if the proper amount of credit will be deducted from the 8-bit register by asserting one of every product signal. First I asserted an apple to get $0.91 ($1.66 – $0.75). Next, I asserted a banana to get $0.71 ($0.91 – $0.20). Next, I asserted a carrot to get $0.41 ($0.71 - $0.30). Lastly, I asserted a date to get $0.01 ($0.41 - $0.40). After I asserted all product signals and obtained expected results, I asserted a reset signal to obtain $0.00 since the reset signal will reset the register back to 00000000 ($0.00). Not shown on the waveform is the test for over-inserting coins past $2.55. When this happens, the credit will just become $2.55. I did not test for negative values because the purchaseMngr checks to see if credit will become negative and outputs an error signal. Below is the waveform for piggyBank.



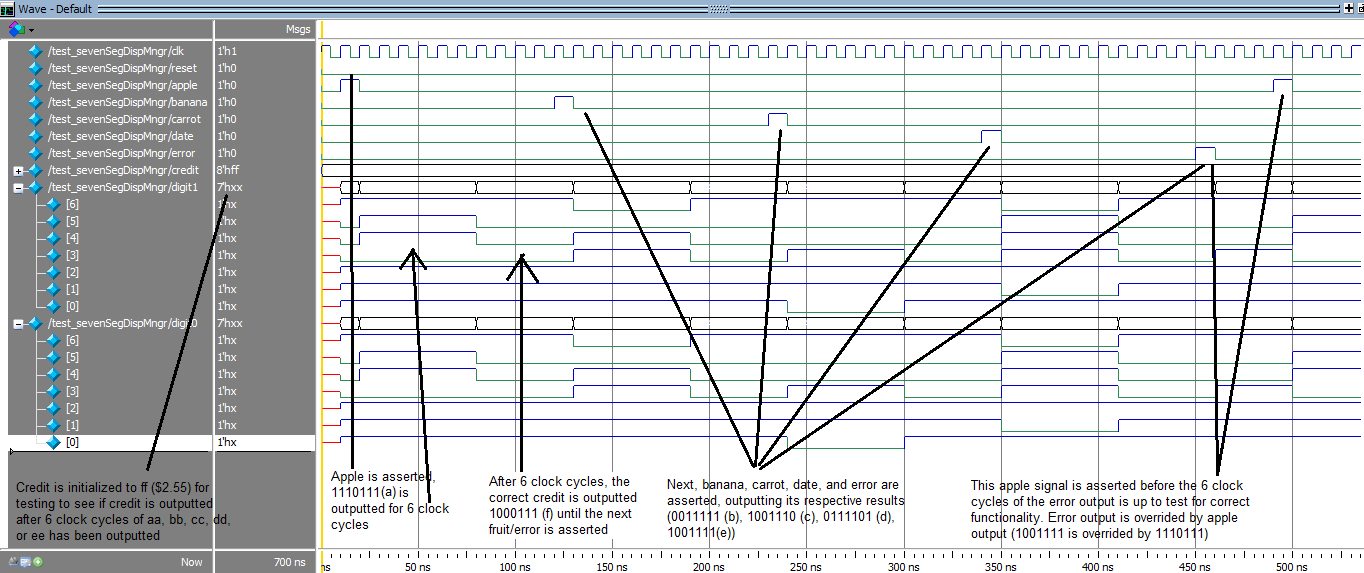
purchaseMngr:

To test purchaseMngr, I first initialized credit to be $2.55 (11111111) to test for normal product code assertions. I asserted a buy signal asserted each combination of product codes to get the appropriate output. Next, I set the credit to be $0.01 (00000001) to test for negative value interactions. After setting credit to $0.01, I asserted a buy signal and an apple signal. Since $0.75 > $0.01, an error signal is outputted. When buy is not asserted, nothing will be outputted even if a product code is present. Below is the waveform for purchaseMngr.



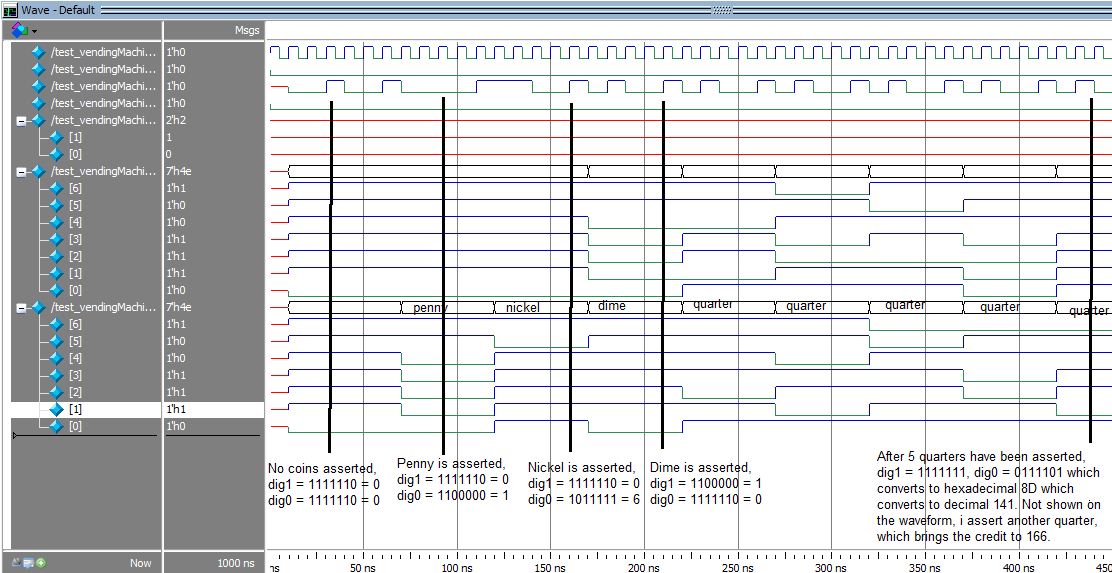
sevenSegDispMngr:

To test the sevenSegDispMngr, I first initialized the credit to be $2.55 (displays “ff”). I first asserted each product/error signal (apple, banana, carrot, date, error) to check for normal behavior. I asserted each signal within 10 clock cycles of each other to test for behavior after the 6 clock cycles of the product letters have been displayed. As expected, after assertion of a product signal, the product letters are outputted for 6 clock cycles. Then, the credit (which is permanently “ff” in this test case) is displayed for 4 clock cycles. After testing the error signal, I did not wait for the 6 clock cycles to finish before asserting the next signal (which is apple). I did this to test for behavior of the module when signals are asserted quickly after another. As wanted, if a product/error signal is asserted before the previous 6 clock cycles of the previous signal is finished, the output will just change to accommodate the most recent product/error signal asserted. Not shown on the waveform is the assertion of a reset signal, which outputs the current credit of the piggyBank.

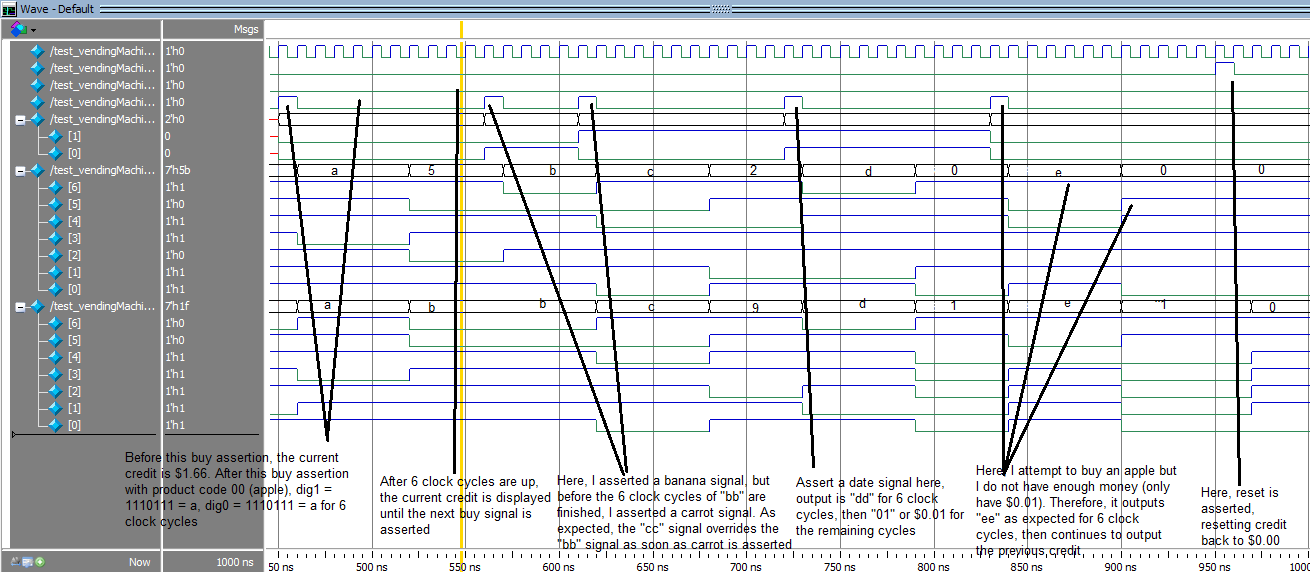


vendingMachine:

To test the full circuit, or vendingMachine module, I first inserted a penny, a nickel, a dime, and 6 quarters to bring the credit to $1.66. Below is the first part of the waveform for vendingMachine. This tests for normalized behavior of coin insertion.



Next, I started making purchases to test for the behavior of purchases. Each purchase is within 10 clock cycles of each other. 6 of those clock cycles should be the product letters being displayed, while the remaining 4 clock cycles should display the current credit. First, I buy an apple and it displays “aa” for 6 clock cycles as expected. After the 6 clock cycles are up, the module displays “5b” which is hexadecimal for 91 or 91 cents, which is the correct output after making an apple purchase ($1.66-$0.75 = $0.91). After, I asserted a banana signal and it displays “bb” correctly. However, to test for behavior of immediately purchasing an item while still within the 6 clock cycles of displaying the previous product letters, I asserted a carrot signal and it immediately displays “cc” without waiting for the 6 clock cycles of “bb” to finish. It then displays the current credit, which is hexadecimal “29” or $0.41 which is the correct output after making a banana and a carrot purchase ($0.91 - $0.20 - $0.30 = $0.41). Next, I purchase a date as normal, and the output is hexadecimal “01” or $0.01 which is correct after making a date purchase ($0.41 - $0.40 = $0.01). Lastly, with only 1 cent stored in piggy bank, I attempt to make an apple purchase. Since $0.75 > $0.01, an error is displayed “ee” for 6 clock cycles, and then the credit is displayed for the remaining clock cycles which is hexadecimal “01” or $0.01. Lastly, I assert a reset signal to test for functionality of reset. As expected, the credit becomes $0.00. Below is part 2 of the waveform of vendingMachine.



**Conclusion**:

In this homework, I have successfully implemented the correct functionality for each module. I broke down the verification of the whole circuit into verification of its individual components first. After testing for normalized use and all corner cases for each individual module, I put each component together into module vendingMachine. From there, I verified the final module by exhausting almost all the corner cases.

With my implementation, there is only one thing that might possibly go wrong and that I have not tested because testing this would require me to change the functionality of my circuit. This situation arises when a coin and a buy/product code signal is asserted within the same clock cycle. Because my circuit is based on an asynchronous design, asserting a coin and a buy/product code signal at the same time might not work correctly. Since I did not have enough time to change all my modules to a synchronous design, this specific corner case is neglected.